Einblicke in die Informatik-Forschung im Bereich Sicherheit

Optimierende Compiler: Vertrauen ist gut, Verifikation ist besser!

Sabine Glesner

Fakultät für Informatik, Universität Karlsruhe

Nicht nur die Rufe nach sicheren Informationssystemen werden immer lauter, auch die Forschungsaktivitäten in diesem Bereich nehmen beständig zu. Um wirklich sichere Informationssysteme zu entwerfen und zu implementieren, müssen viele einzelne Problemstellungen gelöst werden. Ein wichtiger Aspekt untersucht die Sicherheit und Korrektheit von den bei Entwurf und Implementierung eingesetzten Werkzeugen. Compiler sind ein solches Werkzeug. Sie transformieren Programme in menschenlesbarer Notation in ausführbaren Code und erfüllen damit eine zentrale Aufgabe bei der Konstruktion von Softwaresystemen. Auch wenn wir typischerweise darauf vertrauen, dass Compiler korrekt arbeiten, so haben sie doch ihre Fehler, insbesondere optimierende Compiler. In diesem Artikel stelle ich die Forschungsarbeiten meiner Arbeitsgruppe zur Verifikation optimierender Compiler dar und zeige dabei nicht nur beispielhaft typische Probleme, Lösungsansätze und Methoden von Forschungsarbeiten im Bereich Sicherheit auf, sondern diskutiere auch, welche weiterführenden Herausforderungen und Visionen sich daraus ergeben.

Schlüsselworte: Sicherheit (*Safety*), Verifikation, Compiler, Theorembeweiser

1 Einleitung

Die Korrektheit von Compilern ist notwendig, um die Korrektheit damit übersetzter Software sicherzustellen. Auch wenn man im allgemeinen Compilern vertraut, haben diese Software-Werkzeuge dennoch ihre Fehler, wie die Fehlermeldungen gängiger Compiler [Bor02] oder z.B. der in [New01] diskutierte, besonders augenfällige Compilerfehler demonstrieren. Wer kennt nicht das Phänomen, dass man verzweifelt Fehler im eigenen Programm sucht und diese Fehler verschwinden, sobald man Optimierungsstufen im Compiler ausschaltet?

Der vorliegende Artikel verfolgt nicht das Ziel, einen repräsentativen Überblick über die verschiedenen Arbeits- und Forschungsgebiete im Bereich Sicherheit zu geben. Stattdessen stelle ich hier die Forschungsarbeiten meiner Arbeitsgruppe zur Verifikation optimierender Compiler vor und zeige damit beispielhaft, wie typische Fragestellungen und Vorgehensweisen bei der Verifikation von Softwaresystemen (Sicherheit im Sinne von *Safety*) aussehen und welche zukünftigen wissenschaftlichen Herausforderungen sich daraus ergeben. Compiler erfüllen eine zentrale Aufgabe in der Informatik, da sie Programme höherer Programmiersprachen in ablauffähigen Maschinencode transformieren und damit die Lücke schließen zwischen dem, was der Mensch versteht und programmiert und dem, was ein Computer tatsächlich ausführt. Zur Konstruktion sicherer Software sind korrekte Compiler daher eine unabdingbare Voraussetzung. Verifikation verfolgt das Ziel, die korrekte Funktionsweise von Systemen (in unserem Fall von

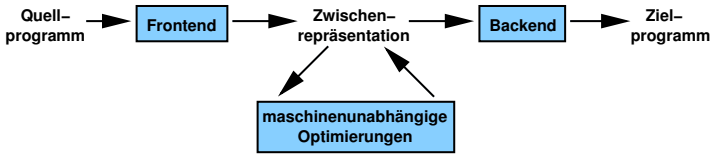


Abbildung 1: Struktur von Compilern

Compilern) zu beweisen. Dies ist gerade bei optimierenden Compilern, die möglichst effizienten Maschinencode erzeugen, eine schwierige Aufgabe. In diesem Artikel zeige ich, welche Lösungen wir für dieses Problem entwickelt haben. Zunächst werden die zentralen Begriffe *Compiler* (Abschnitt 2) und *Verifikation* (Abschnitt 3) eingeführt. Anschließend wird im Abschnitt 4 der Stand von Forschung und Technik in Bezug auf Korrektheit von Compilern dargestellt. Außerdem werden die Forschungsfragen, die sich bei der *Verifikation optimierender Compiler* ergeben, erläutert. Weiterhin stelle ich in den Abschnitten 5 und 6 meine Forschungsarbeiten zur Korrektheit und Sicherheit optimierender Compiler vor. In Abschnitt 7 werden verwandte Arbeiten erläutert. Abschließend, in Abschnitt 8, fasse ich die Ergebnisse zusammen und zeige zukünftige Forschungsperspektiven auf.

2 Compiler

Compiler (auch *Übersetzer* genannt) transformieren Programme von einer Programmiersprache in eine andere Programmiersprache, also zum Beispiel Programme einer höheren Programmiersprache wie C, Java, Lisp oder Pascal in ausführbaren Maschinencode, d.h. in die 0-1-Folgen, die ein Prozessor letztendlich verarbeitet. Das Gebiet des Compilerbaus wird oft als eine "Perle der Informatik" bezeichnet, weil viele der darin auftretenden Probleme algorithmisch sehr gut verstanden sind und weil man außerdem auch die Systemarchitektur von Compilern gut im Griff hat, d.h. es gibt bewährte Ingenieurmethoden, mit denen man den praktikablen Teil der Theorie in effiziente, auch für die industrielle Praxis geeignete Systeme umsetzen kann [ASU92]. Die Struktur von Compilern ist überblicksartig in Abbildung 1 dargestellt: Die Eingabe eines Compilers ist ein Quellprogramm, das zunächst im Frontend in eine Zwischenrepräsentation transformiert wird. Diese Zwischenrepräsentation stellt die strukturellen Eigenschaften und insbesondere den Daten- und Steuerfluss von Programmen dar. Während der Datenfluss anzeigt, welche Werte an welchen Stellen im Programm berechnet bzw. für weitere Berechnungen verwendet werden, spezifiziert der Steuerfluss, welche Teile des Programms in welcher Reihenfolge eventuell auch in Abhängigkeit vom Datenfluss ausgeführt werden. Die Zwischenrepräsentation kann durch maschinenunabhängige Optimierungen verbessert werden (zum Beispiel indem nie ausgeführte Programmteile eliminiert werden), bis schließlich im Backend das Zielprogramm erzeugt wird.

Für die Frontendaufgaben von Compilern gibt es sehr zufriedenstellende Lösungen. Programmiersprachen werden mit regulären und kontextfreien Sprachen sowie mit attributierten Grammatiken oder verwandten Formalismen (zum Beispiel natürliche Semantik) beschrieben. Für jede dieser Methoden gibt es effiziente Prüfalgorithmen, die entscheiden, ob ein vom Programmierer geliefertes Eingabeprogramm tatsächlich gültig ist und damit zur Programmiersprache gehört. Frontends können relativ leicht implementiert werden, weil es Generatoren (Computerprogramme) gibt, mit denen sie aus geeigneten Spezifikationen automatisch erzeugt werden können. Bekannte Generatoren sind z.B. die Unix-Werkzeuge Lex (Lexical Analyzer Generator)

und Yacc (Yet Another Compiler Compiler), mit denen Scanner und Parser zur Durchführung der lexikalischen und syntaktischen Analyse generiert werden können.

Im Bereich der maschinenunabhängigen Optimierungen und der Backend-Transformationen [Muc97] gibt es noch viele offene Forschungsprobleme, die nicht nur theoretisch interessante Fragestellungen aufwerfen, sondern auch aus praktischer Sicht dringend gelöst werden müssen. Die maschinenunabhängigen Optimierungen analysieren den Daten- und Steuerfluss in Programmen und nutzen die dabei gewonnenen Informationen, um Programme gewinnbringend zu transformieren [NNH99], ohne bereits die Architektur einer bestimmten Zielmaschine zu berücksichtigen. Zum Beispiel könnte eine Analyse untersuchen, ob es in einem Programm identische Teilergebnisse gibt, die mehrfach berechnet werden (weil sie an verschiedenen Programmstellen benötigt werden), die man aber auch genauso gut nur einmal berechnen könnte, wodurch sich das Zeitverhalten des Programms verbessern ließe. Ähnliche Methoden wurden zum Beispiel vor dem Jahrtausendwechsel erfolgreich eingesetzt, um Software für diesen bedeutsamen Jahreswechsel fit zu machen. Man hatte hier nämlich das Problem, dass sehr viel alte Software in Benutzung war und ist, bei deren Entwicklung die fatale Fehlentscheidung getroffen worden war, Jahreszahlen mit zweistelligen Zahlen zu kodieren, also etwa 71 für das Jahr 1971. Wenn nun Jahreszahlen auf einmal kleiner werden, zum Beispiel 00 im Jahr 2000, dann können unerwartete Ergebnisse entstehen (zum Beispiel verlieren Kreditkarten ihre Gültigkeit, wenn ihr Ablaufdatum scheinbar kleiner als das aktuelle Datum ist) und zu fehlerhaftem Verhalten der Software führen. Die Entscheidung, Jahreszahlen verkürzt darzustellen, war vor einigen Jahrzehnten zur Entstehungszeit der Software getroffen worden, als Speicherplatz äußerst knapp war und nicht mit vierstelligen Jahreszahlen verschwendet werden sollte. Um katastrophale Auswirkungen dieser kurzsichtigen Entscheidung zu verhindern, hat man in den 90er Jahren Software erfolgreich daraufhin analysiert und modifiziert. Es ist eine der großen Leistungen der Informatik, dass der Jahrtausendwechsel so problemlos stattfand und gleichzeitig ein Beispiel dafür, dass Methoden aus dem Compilerbau auch in anderen Bereichen der Informatik bei der Transformation und Optimierung von Systemen einsetzbar sind.

Auch der Bereich der Backends von Compilern ist ein aktives Forschungsgebiet. Codegenerierung ist allgemein ein sehr schweres Problem. Die Informatik klassifiziert in der Komplexitätstheorie [Pap94] Probleme bzgl. ihrer prinzipiellen Berechnungszeit, die man für ihre Lösung benötigt (zum Beispiel konstant, linear, quadratisch, polynomiell oder gar exponentiell anwachsenden Zeitaufwand). Eine wichtige Klasse von Problemen sind **NP**-vollständige Probleme (**NP**=nichtdeterministisch polynomiell), für die es keine bekannten, für die Praxis ausreichend effizienten Lösungsverfahren gibt (lediglich exponentiell anwachsende und damit praktisch nicht verwendbare Suchverfahren sind bekannt), für die man aber die Korrektheit von Lösungen effizient (=polynomiell) prüfen kann. Im Compilerbau lebt man schon seit langem mit der Situation, dass man **NP**-vollständige Probleme effizient lösen muss und hat dabei sehr gute Ingenieurmethoden entwickelt. Codegenerierung ist ein Beispiel dafür. Antatt immer die optimale Lösung zu suchen, macht man Abstriche und ist auch mit ausreichend guten Ergebnissen zufrieden.

Codegenerierung ist insbesondere für neuere Prozessorarchitekturen eine wissenschaftliche Herausforderung. In den letzten Jahrzehnten konnte man beobachten, dass sich die Hardwareleistung alle paar Jahre verdoppelt hat. Dieser Trend kann nur durch Hardwareverbesserungen allein nicht fortgesetzt werden, weil man an physikalische Grenzen stößt. Daher ist eine verstärkte Interaktion zwischen Compiler und Hardware gefordert. Nur wenn Compiler besser optimieren und die Möglichkeiten der Hardware besser nutzen, können Prozessoren ihr Leistungspotential voll entfalten. Das ist besonders in eingebetteten Systemen (z.B. in Automobil- oder Multimediasystemen) sehr wichtig, weil hier oft strikte Randbedingungen bzgl. Berech-

nungszeit, Speicherplatz und Produktionskosten gelten und die Prozessoren in diesen Systemen irreguläre Strukturen aufweisen, die diese Optimierungsaufgabe erschweren.

Insgesamt präsentiert sich der Compilerbau heute als ein Bereich, in dem bereits wesentliche Ergebnisse erzielt worden sind, der sich aber nach wie vor als ein aktives Forschungsgebiet mit sowohl theoretisch schwierigen als auch aus praktischer Sicht dringenden Herausforderungen darstellt.

3 Verifikation

Korrekte Software zu erstellen ist bereits seit den Anfängen der Informatik ein wichtiges Forschungsziel und hat seitdem nicht an Aktualität verloren. Als relativ aktuelles Beispiel denke man nur an den "Pentium-Bug", der 1994 bei bestimmten Gleitkommadivisionen im Pentium-Prozessor ein falsches Ergebnis geliefert und Intel einen Schaden in Höhe von knapp 500 Millionen Dollar verursacht hat. Als Konsequenz aus diesem Verlust betreibt Intel heute die formale Verifikation von Gleitkommaalgorithmen [Har00], eines von vielen Beispielen dafür, dass monetäre Interessen den Einsatz formaler Methoden durchaus erzwingen können, sobald Sicherheit und Zuverlässigkeit zu entscheidenden Wettbewerbsfaktoren werden.

Verifikation bedeutet, dass man einen mathematischen Beweis führt, dass ein System, zum Beispiel ein Software-System, das tut, was man von ihm erwartet. Dabei werden insbesondere Methoden aus der formalen Logik eingesetzt, die ursprünglich ein Zweig der reinen Mathematik und inzwischen auch ein wichtiges Forschungsgebiet der theoretischen Informatik ist. Dieser Zusammenhang zur Logik beruht auf der Beobachtung, dass alle Berechnungen, die ein Rechner ausführt, letztendlich rein syntaktische Manipulationen sind; sei es auf Ebene höherer Programmiersprachen oder auf Ebene der 0-1-Folgen, die ein Prozessor verarbeitet. Alle solchen Zeichenfolgen haben a priori im Computer keine Bedeutung und werden lediglich nach festen Schemata bearbeitet. Die Bedeutungen, d.h. die *Semantik*, die wir den Ein- und Ausgaben eines Softwaresystems geben, müssen einigen formalen Regeln genügen und können in gewissen Grenzen variiert werden.

Beispiel (Semantik) *Zur Erläuterung des Semantikbegriffs möchte ich ein kleines Märchen erzählen: Eine Prinzessin wurde von einem bösen Zauberer entführt, der ihr zwar Lesen und Schreiben beibrachte, aber dabei die Reihenfolge des Alphabets umdrehte, ihr also das "A" als "Z" vorstellte, das "B" als "Y" usw.:*

"normale Buchstaben"	**A**	**B**	*C*	*D*	**E**	*F*	...	*Q*	**R**	*S*	*T*	*U*	*V*	*W*	*X*	*Y*	*Z*
Buchstaben der Prinzessin	**Z**	**Y**	*X*	*W*	**V**	*U*	...	*J*	**I**	*H*	*G*	*F*	*E*	*D*	*C*	*B*	*A*

Die Prinzessin konnte mit ihrer Bedeutung, ihrer **Semantik** *der Buchstabenzeichen perfekt lesen und schreiben, auch wenn die Wörter völlig anders aussahen als im Rest des Königreichs. Zum Beispiel schrieb die Prinzessin das Wort* **RABE** *in ihrer Schrift als* **IZYV**. *Da der Zauberer ihr nur Texte zum Lesen überließ, in der er die Buchstaben gemäß obiger Tabelle ausgetauscht hatte, fiel der Prinzessin nichts ungewöhnliches auf. Eines Tages kam ein Prinz in die Gegend und durfte, wie sich das für ein Märchen gehört, nicht mit der Prinzessin reden. Aber glücklicherweise war ein Rabe so nett, schriftliche Botschaften zwischen ihr und dem Prinzen auszutauschen. Leider konnten die beiden Verliebten ihre Nachrichten gegenseitig nicht verstehen, weil sie beide eine unterschiedliche* **Semantik** *für die Buchstaben hatten, und waren bald ganz verzweifelt. Wieder erwies sich der Rabe als rettender Engel, denn er hatte ein Selbstgespräch des Zauberers belauscht, in dem dieser verraten hatte, wie er die Prinzessin so erfolgreich verwirren konnte. Der Rabe verriet dem Prinzen die Bedeutung, die* **Semantik** *der Botschaften,*

die die Geliebte ihm geschickt hatte, und bald wurde im Königreich ein Hochzeitsfest gefeiert.

In der Mathematik und Informatik werden unterschiedliche Semantiken betrachtet. In der Mathematik ist man an Semantiken interessiert, die mathematischen Aussagen einen Wahrheitswert (wahr oder falsch) zuweisen. Im klassischen Ansatz ist jede mathematische Aussage entweder wahr oder falsch. Man weiß zwar manchmal nicht, ob ein bestimmter Satz tatsächlich wahr oder falsch ist, aber eine der beiden Alternativen muss gelten ("tertium non datur"*). In der konstruktiven Logik dagegen kann man nur solche Formeln beweisen, deren Beweise gleichzeitig Programme sind, mit denen man gültige Instantiierungen der bewiesenen Formel berechnen kann. In der Informatik betrachtet man nicht nur Semantiken, die Wahrheitswerte definieren, sondern insbesondere bei der Software-Verifikation solche Semantiken, mit denen man die Bedeutung von Programmen definiert. In der denotationellen Semantik zum Beispiel ordnet man jedem Programm eine mathematische Funktion zu, die die Transformation der Programmeingaben in seine Ausgaben beschreibt. Ausführlichere Semantiken beschreiben nicht nur das Ein-Ausgabe-Verhalten, sondern auch die Zustandsfolgen, die bei der Ausführung von Programmen durchlaufen werden. Mit derartigen zustandsorientierten Semantiken kann man auch die Semantik nichtterminierender Programme (wie zum Beispiel Betriebssysteme, Datenbanken, reaktive Systeme) beschreiben und Aussagen über ihr Verhalten ableiten.* ◇

Bei der Verifikation muss man zuerst festlegen, was man überhaupt von einem System erwartet. Dazu gibt man eine Spezifikation an, in der definiert ist, welche Eingaben zulässig sind und wie sich das System in Abhängigkeit seiner Eingaben verhält. Außerdem benötigt man einen Kalkül, der wahre Grundannahmen, die *Axiome*, sowie *Inferenzregeln* enthält, mit denen man aus bereits vorhandenen wahren Aussagen weitere wahre Aussagen ableiten kann. Bei der Verifikation von (Software-)Systemen besteht eine solche Aussage darin, dass ein System bzgl. seiner Spezifikation korrekt ist, d.h. sich so verhält wie in der Spezifikation gefordert. Schließlich muss man zeigen, dass der Kalkül bzgl. der Semantik der betrachteten Systeme korrekt ist, dass also alle Aussagen, die damit abgeleitet werden können, auch bzgl. der damit intendierten Bedeutung korrekt sind. Zum Abschluss muss man in dem Kalkül die Aussage ableiten, dass das System bzgl. seiner Spezifikation korrekt ist.

Verifikation kann mit maschineller Unterstützung durchgeführt werden, denn Kalküle sind formale Systeme, die in einem Computer implementiert werden können. In meiner Arbeitsgruppe verwenden wir den Theorembeweiser Isabelle [NPW02], und es gibt eine Reihe weiterer Beweissysteme, die jeweils ihre eigenen Stärken haben. Gerade bei der Verifikation großer Software-Systeme sind viele ähnliche, aber nicht identische Fallunterscheidungen nötig, so dass die Unterstützung durch einen Theorembeweiser zum einen dabei hilft, den Nachweis dieser vielen ähnlichen Beweisschritte zu automatisieren, und zum anderen sicherstellt, dass man keinen Fall übersehen hat und der Beweis wirklich vollständig ist. Allerdings muss man zur Kenntnis nehmen, dass die Verifikationstechnologie heutzutage noch sehr teuer ist, weil viele Beweisschritte nicht automatisierbar sind und menschliche Kreativität erfordern. Simulation bzw. Testen als billigere Alternative zur Verifikation bietet sich insbesondere in sicherheitskritischen Bereichen kaum an, weil man damit nicht sicherstellen kann, dass ein System für **alle** möglicherweise auftretenden Situationen korrekt arbeitet. Eine solche Allaussage kann man nur mit einem echten Beweis formal herleiten.

Inzwischen gibt es zahlreiche erfolgreiche Projekte, die sich mit der Spezifikation und formalen Verifikation von Software beschäftigen, zum Beispiel die Formalisierung eines Teils der Programmiersprache Java [NvO98] oder die Fallstudien, in denen Anforderungen an die Software in Space Shuttles der NASA formalisiert wurden und zu Verbesserungen der Software geführt haben [CV98] oder die Verifikation des Protokolls, das den Verkehr durch den Elbtun-

nel in Hamburg steuert [OST+02], um nur eine kleine Auswahl zu nennen. Allerdings dürfen diese Erfolge nicht darüber hingweg täuschen, dass Softwareverifikation momentan mehr als eine Kunst denn als eine Ingenieuraufgabe angesehen wird. Es ist eine der großen Aufgaben der Informatik, automatisierbare Verfahren zu entwickeln, mit denen korrekte und verlässliche Systeme konstruiert und gewartet werden können.

4 Korrektheit von Compilern: Stand der Technik

Compiler sind das Herzstück bei der Konstruktion und Wartung von Software, erlauben sie es doch, Programme in höheren Programmiersprachen zu schreiben, die dann mit Hilfe von Compilern in ausführbaren Maschinencode transformiert werden. Um zuverlässige Software zu erstellen, ist es daher unbedingt erforderlich, dass Compiler nachweislich korrekt arbeiten.

Bei der Korrektheit von Compilern unterscheidet man zwei verschiedene Fragestellungen [GGZ04]: Zum einen untersucht man, ob ein gegebener Übersetzungsalgorithmus korrekt ist, d.h. ob er die Bedeutung, die *Semantik* der transformierten Programme erhält. Zum anderen stellt man die Frage, ob ein eventuell zuvor verifizierter Übersetzungsalgorithmus in einem vorliegenden Compiler auch korrekt implementiert ist. Den ersten Korrektheitsbegriff, der sich auf die semantische Korrektheit der Übersetzungsalgorithmen bezieht, bezeichnet man mit *Übersetzungskorrektheit*, den zweiten, der die Korrektheit der Implementierungen in Compilern betrachtet, mit *Implementierungskorrektheit*. Übersetzungskorrektheit wurde erstmalig bei der Übersetzung arithmetischer Ausdrücke [MP67] betrachtet. Implementierungskorrektheit wurde in [Pol81, CM86] zum ersten Mal untersucht. Im Folgenden stellen wir den Stand von Forschung und Technik bezüglich dieser beiden Begriffe dar.

4.1 Übersetzungskorrektheit

Bei dem Nachweis der Übersetzungskorrektheit untersucht man Algorithmen, die Programme transformieren. Das sind zum einen Algorithmen, die Programme von einer Programmiersprache in eine andere übersetzen. Zum anderen sind es Algorithmen, die Programme modifizieren und dadurch optimieren (siehe auch die Erläuterungen zu maschinenunabhängigen Transformationen in Abschnitt 2). Bei der Verifikation von Transformationsalgorithmen muss man nachweisen, dass die Semantik der transformierten Programme, die typischerweise das Ein-Ausgabe-Verhalten der Programme beschreibt, erhalten bleibt. Je nach Art der Transformationsalgorithmen sind dazu unterschiedliche Beweismethoden erforderlich.

In bisherigen Arbeiten werden meist Verifikationen von Verfeinerungstransformationen betrachtet. Das sind solche Übersetzungen, bei denen die Struktur der Programme nicht verändert wird, sondern lediglich während der Übersetzung immer genauer festgelegt wird, wie die einzelnen Berechnungen auszuführen sind. Zum Beispiel würde man bei der Übersetzung höherer Programmiersprachen in Maschinencode bestimmen, wie komplexe Datenstrukturen in die Speicherhierarchie des Prozessors abgebildet werden. Das bedeutet insbesondere, dass Programme nach einem *Divide et Impera*-Prinzip lokal übersetzt und anschließend die Übersetzungen wieder zusammengefügt werden können. Die entsprechenden Korrektheitsbeweise folgen diesem Prinzip: Korrektheit kann lokal gezeigt werden, und globale Korrektheit folgt daraus. Verifikationen nach diesem Schema finden sich unter anderem in [DvHG03, SA97].

4.2 Implementierungskorrektheit

Die Frage der Implementierungskorrektheit von Compilern ist aus softwaretechnischer Sicht von großer Bedeutung, insbesondere wenn man bedenkt, dass Compiler heutzutage nicht von Hand

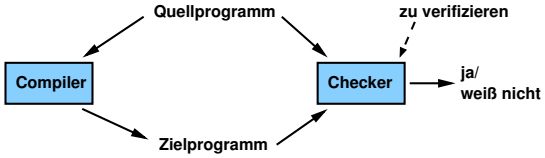


Abbildung 2: Compiler mit Checker

geschrieben, sondern aus geeigneten Spezifikationen mittels Generatoren erzeugt werden. Man könnte nun versuchen, diese Generatoren selbst zu verifizieren. Angesichts der Größe dieser Systeme entfällt diese Option, insbesondere wenn man maschinelle Beweise in Theorembeweisern führen möchte. Mit heute verfügbaren Verifikationsmethoden kann man Software dieser Größenordnung (noch) nicht in den Griff bekommen. Man könnte stattdessen versuchen, die generierten Compiler zu verifizieren. Auch diese Möglichkeit entfällt, mit derselben Begründung.

Als Ausweg aus diesem Dilemma hat sich in den letzten Jahren Programmprüfung als die Methode der Wahl etabliert, in der Literatur auch als *translation validation* [PSS98] oder *program checking* [GGZ98] bekannt. Anstatt den Compiler zu verifizieren, verifiziert man nur sein Ergebnis. Dabei geht man so vor, dass man den Compiler mit einem unabhängigen Checker anreichert, vergleiche Abbildung 2. Der Checker erhält jeweils das Quellprogramm und das daraus generierte Zielprogramm als Eingabe und überprüft, ob die beiden Programme semantisch äquivalent sind. Im Falle einer positiven Entscheidung haben wir einen Nachweis, dass tatsächlich eine korrekte Übersetzung stattgefunden hat. Im negativen Fall weiß man gar nichts. Aus theoretischer Sicht kann man nicht erwarten, dass der Checker immer entscheiden kann, ob eine korrekte Transformation stattgefunden hat, weil Programmäquivalenz ein unentscheidbares Problem ist und deshalb nicht algorithmisch für beliebige Eingaben gelöst werden kann. Aus praktischer Sicht hat sich dieses Vorgehen aber als sehr gut geeignet erwiesen. Wenn man weiterhin den Checker bzgl. einer geeigneten Spezifikation formal verifiziert, dann erhält man im Fall einer positiven Entscheidung des Checkers ein formal verifiziertes Übersetzungsergebnis. Diese Methode zum Nachweis der Implementierungskorrektheit hat sich in allen Phasen der Frontends von Compilern (lexikalische, syntaktische und semantische Analyse) als sehr gut geeignet erwiesen, insbesondere auch deshalb, weil Ergebnisse in diesen Phasen der Übersetzung eindeutig sind [HGG+99, Gle03b, GFJ04]. Man kann sich diese Prüfmethode wie den *Test durch Probe* vorstellen, den man in der Grundschule kennen gelernt hat, um die Ergebnisse von schriftlichen Divisionsaufgaben zu prüfen: Ist man sich nicht sicher, ob das errechnete Ergebnis einer solchen Division richtig ist, multipliziert man einfach den Divisor mit dem Ergebnis und vergleicht mit dem Dividenden. Stimmen sie überein, hat man keinen Fehler gemacht. So funktioniert auch Programmprüfung. Man verwendet im Checker Tests, die wesentlich einfacher durchzuführen sind als die eigentliche Berechnung der Lösung.

4.3 Ungelöste Probleme bei der Verifikation optimierender Compiler

Bislang wurde nicht geklärt, wie die Übersetzungskorrektheit von Transformationen nachgewiesen werden kann, die auch die Struktur von Programmen verändern können. Dieses Problem behandeln wir in Abschnitt 5. Außerdem haben bisherige Arbeiten nicht geklärt, wie für optimierende Transformationen, bei denen es mehrere, eventuell sogar viele korrekte Lösungen gibt, Implementierungskorrektheit sichergestellt werden kann. In Abschnitt 6 stellen wir eine Lösung dafür vor.

5 Übersetzungskorrektheit optimierender Compiler

In meinen Arbeiten (insbesondere in [Gle04c]) konzentriere ich mich auf die Verifikation der Transformationen in Compilern, die stattfinden, sobald ein Programm in der internen Zwischenrepräsentation vorliegt. Da diese Transformationen besonders fehleranfällig sind, ist ihre Verifikation aus praktischer Sicht äußerst wichtig. Gleichzeitig sind diese Verifikationen auch aus theoretischer Sicht sehr interessant, weil sie neue semantische Konzepte erfordern.

5.1 Semantik nichtterminierender Programme

Klassischerweise konzentriert man sich auf *terminierende* Programme, die irgendwann ihre Berechnungen abschließen und zu einem Ende kommen. Die Semantik terminierender Programme kann durch das Ergebnis ihrer Berechnungen oder, falls man einen genaueren Blick auf das Programmverhalten benötigt, durch die endliche Sequenz der Zustände beschrieben werden, die bei ihrer Abarbeitung durchlaufen werden. Bei der Verifikation von Programmtransformationen muss man entsprechend nachweisen, dass das Ergebnis der Berechnung bzw. die endliche Sequenz durchlaufener Zustände erhalten bleibt. Dazu geeignete Beweismethoden kombinieren das im Abschnitt 4.1 beschriebene *Divide et Impera*-Prinzip mit einer Induktion über die Anzahl durchlaufener Zustände. Derartige Beweismethoden können allerdings auf unendliche Zustandssequenzen nicht angewendet werden. Unendliche Zustandssequenzen sind jedoch notwendig, um die Semantik nichtterminierender Programme adäquat zu beschreiben.

Um diese Einschränkung zu überwinden, arbeite ich an Semantiken, mit denen auch nichtterminierende Programme beschrieben und ihre Transformationen verifiziert werden können. Das ist insbesondere für die Praxis wichtig, denn man findet nichtterminierende Programme in einer Vielzahl von sicherheitskritischen Anwendungen, wie zum Beispiel in Betriebssystemen und Datenbanken sowie in reaktiven und eingebetteten Systemen. In meinen Arbeiten [Gle04a] habe ich gezeigt, dass Semantik *koinduktiv* definiert werden kann und dass damit auch Transformationen nichtterminierender Programme verifiziert werden können. *Koinduktion* [JR97] ist das zur Induktion duale Definitions- und Beweisprinzip, das sich insbesondere zur Beschreibung und Verifikation zustandsbasierter Systeme, wozu auch Programme gehören, eignet.

5.2 Maschineller Nachweis der Übersetzungskorrektheit

Ein Schwerpunkt unserer Verifikationsarbeiten besteht darin, die Übersetzungskorrektheit von Transformationsalgorithmen maschinell mit Hilfe von Theorembeweisern nachzuweisen (vergleiche auch die Diskussion der entsprechenden Vor- und Nachteile im Abschnitt 3). In diesem Abschnitt fasse ich die Ergebnisse unserer Arbeiten zusammen, in denen wir die Codeerzeugung in Compilern betrachtet und für diese Phase die Übersetzungskorrektheit maschinell im Theorembeweiser Isabelle/HOL nachgewiesen haben.

Ausgangspunkt unserer Untersuchungen ist eine SSA(*static single assignment*)-basierte Zwischenrepräsentation[1]. Diese Repräsentationsform stellt die essentiellen Datenabhängigkeiten in Programmen direkt dar und ermöglicht so Optimierungen besonders gut. Wie die meisten Zwischensprachen in Compilern sind auch SSA-Sprachen grundblockorientiert, d.h. maximale Sequenzen verzweigungsfreier Anweisungen werden in Grundblöcken angeordnet, und der Steuerfluss verbindet die Grundblöcke miteinander. Zudem wird in SSA-Darstellung jeder Variablen statisch nur einmal ein Wert zugewiesen, eine Darstellung, die man unter anderem durch geeignete Duplizierung und Umbenennung der Variablen immer erreichen kann und wodurch man die essentiellen Datenabhängigkeiten eines Programms besonders gut erkennen kann. Innerhalb

[1] deutsch: SSA = Sprachen mit statischer Einmalzuweisung

von Grundblöcken sind Berechnungen ausschließlich datenfluss-getrieben, d.h. eine Operation kann ausgeführt werden, sobald ihre Eingabewerte festliegen. In [Gle04b] habe ich für SSA-Zwischensprachen eine formale Semantik entwickelt. Die Formalisierung dieser Semantik im Theorembeweiser Isabelle/HOL haben wir in [BG04, GB05] beschrieben, ebenso wie den Nachweis der Korrektheit der Codeerzeugung ausgehend von SSA-Darstellungen.

Bei der Formalisierung in Theorembeweisern existieren oft mehrere Möglichkeiten, wie man Spezifikationen angeben und darauf aufbauend Beweise führen kann. Auch bei der Verifikation der Codeerzeugung gibt es mehrere Beweisalternativen. Wir haben daher zwei verschiedene Formalisierungen erstellt und miteinander verglichen [BGLM05]. Beide Korrektheitsbeweise sind in dem Theorembeweiser Isabelle/HOL geführt worden und zeigen damit beide dasselbe Resultat. Ist man nur an diesem Resultat interessiert, sind beide Beweise gleichwertig, insbesondere auch deshalb, weil sie sich in ihrer Länge kaum unterscheiden. Aus mathematisch-logischer Sicht unterscheiden sie sich jedoch stark. Dahinter steckt eine Erfahrung, die auch Mathematiker bei ihrer Arbeit machen. Es gibt Beweise, die sich "gut" anfühlen, bei denen man die richtige Intuition getroffen hat. Man kann zwar keine formale Definition angeben, wann sich ein gegebener Beweis tatsächlich gut anfühlt, meist aber weiß man es genau, sobald man ihn hat, vergleiche [AZ04]. Diese Erfahrung haben wir auch gemacht. In unserem "guten" Beweis passen die einzelnen Beweisschritte mit unserer intuitiven Beweisidee zusammen und formalisieren ein generelles Prinzip, nämlich dass die Transformation eines Programms die Datenabhängigkeiten erhalten muss. Dadurch, dass wir unseren Beweis auf dem generellen Prinzip "Erhaltung der Datenabhängigkeiten" aufgebaut haben, können wir den Beweis auch bei der Verifikation weiterer Transformationen wiederverwenden. In aktuellen Arbeiten setzen wir ihn zum Beispiel bei der Verifikation der Eliminierung toten Codes und bei der Verifikation von Schleifentransformationen ein, beides typische Optimierungen in modernen Compilern. Viele weitere Anwendungsmöglichkeiten existieren.

Formale Softwareverifikation mit Hilfe von Theorembeweisern wie Isabelle/HOL ist heutzutage aus zwei Gründen möglich: Erstens hat sich die Geschwindigkeit von Prozessoren so gesteigert, dass die komplexen Berechnungen in Theorembeweisern ausreichend schnell durchgeführt werden können. Zweitens hat sich die Benutzerfreundlichkeit von Theorembeweisern so verbessert, dass diese Systeme auch von Arbeitsgruppen verwendet werden, die nicht an ihrer Entwicklung beteiligt waren. Trotzdem ist formale Softwareverifikation sehr teuer und erfordert aufwändige Benutzerinteraktionen. Dieser Aufwand kann reduziert werden, wenn es uns gelingt, Beweise wiederzuverwenden, indem wir Beweise auf allgemeingültigen Prinzipien für die Korrektheit von Systemtransformationen (wie die Erhaltung der Datenabhängigkeiten) aufbauen. Unsere hier vorgestellten Arbeiten zur Übersetzungskorrektheit sind dafür ein Beispiel.

6 Implementierungskorrektheit optimierender Compiler

Bei der Codeerzeugungsphase ist wie bei fast allen Problemen in Backends von Compilern die Optimierungsvariante eines **NP**-vollständigen Problems zu lösen. Damit ist gemeint, dass man nicht nur eine beliebige, sondern auch eine möglichst gute Lösung für ein **NP**-vollständiges Problem sucht. Probleme in **NP** sind dadurch gekennzeichnet, dass ihre Lösungen immer in polynomieller Zeit auf Korrektheit geprüft werden können [Pap94], vergleiche auch Abschnitt 2. Das bedeutet, dass eine nichtdeterministische Turingmaschine zwar einen potentiell exponentiell großen Suchraum durchlaufen muss, bevor sie eine Lösung findet, die Lösung selbst ist aber immer in polynomieller Tiefe zu finden, vergleiche Abbildung 3. Wenn man den Weg zu einer Lösung (in der Komplexitätstheorie auch *Zertifikat* genannt) kennt, dann kann man die Lösung schnell, d.h. in polynomieller Zeit berechnen. Zum Beispiel bei dem Problem, die Erfüllbarkeit

aussagenlogischer Formeln zu entscheiden, enthält das Zertifikat eine erfüllende Belegung.

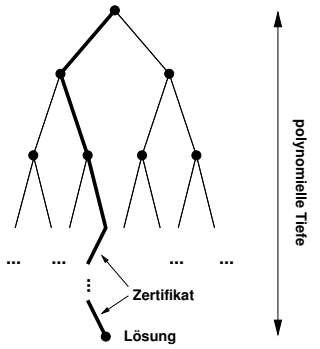


Abbildung 3: **NP**-Berechnungen

Diesen Zusammenhang nutzen wir bei der Untersuchung der Implementierungskorrektheit für die Codeerzeugung aus. Wir modifizieren das bisherige Checker-Szenario so, dass der Compiler neben dem Zielprogramm auch ein Zertifikat erzeugt, in dem festgehalten ist, wie die Lösung berechnet wurde, vergleiche Abbildung 4. Der Checker erhält dieses Zertifikat als dritte Eingabe und verfährt damit wie folgt: Er berechnet aus dem Eingabeprogramm mithilfe des Zertifikats ein Zielprogramm, vergleicht das selbst berechnete Zielprogramm mit dem vom Compiler berechneten und gibt im Fall der Übereinstimmung eine "ja"-Antwort, im negativen Fall dagegen das Ergebnis "weiß nicht". Diese Prüfmethode bezeichnen wir als *Programmprüfen mit Zertifikaten*.

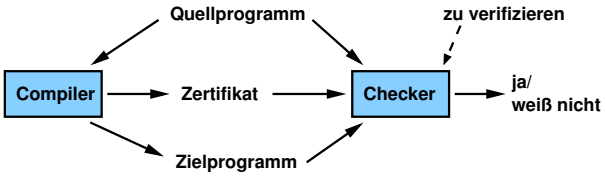


Abbildung 4: Programmprüfen mit Zerifikaten

Man mag sich fragen, was passiert, wenn der Compiler ein fehlerhaftes Zertifikat ausgibt. Wenn der Checker mit solch einem inkorrekten Zertifikat ein korrektes Ergebnis berechnet und wenn weiterhin dieses Ergebnis mit dem vom Compiler berechneten übereinstimmt, dann hat es der Checker geschafft, das vom Compiler berechnete Ergebnis zu verifizieren. Dabei spielt es keine Rolle, wie der Compiler sein Ergebnis ermittelt hat, solange es der Checker mit seiner (verifizierten) Implementierung rekonstruieren konnte.

Betrachten wir noch einmal die Rolle von Codegenerator und Checker genauer. Der Codegenerator verhält sich wie eine nichtdeterministische Turingmaschine, indem er die Lösung sucht und berechnet. Der Checker dagegen agiert wie eine deterministische Turingmaschine und berechnet die Lösung, und zwar auf dieselbe Weise wie der Codegenerator mithilfe des Zertifikats. Das bedeutet, dass man erwarten kann, dass die Implementierung des Checkers mit einem Teil der Implementierung des Codegenerators übereinstimmt, nämlich mit dem Teil, der die Lösung berechnet. Diese Erwartung haben wir in unseren Experimenten bestätigt gefunden. Auf den ersten Blick mag das erstaunlich wirken, weil man ursprünglich Checker verwendet, um einen von der Implementierung unabhängigen Test auf Korrektheit durchführen zu können. Hier hat sich das Prüfparadigma geändert. Wir verwenden den Checker nicht für solch einen unabhängigen Test, sondern als eine Methode, um den korrektheitskritischen Anteil einer Implementierung zu separieren. Die Berechnung der Lösung ist korrektheitskritisch; die Suche nach einer guten Lösung ist es nicht, denn die Güte einer Lösung beeinflusst ihre Korrektheit nicht.

Wir haben mittels der Methode des Programmprüfens mit Zertifikaten einen Programmprüfer für einen Codegenerator in einem industriellen Projekt entworfen und implementiert [Gle03a, Gle03b]. Der betrachtete Codegenerator umfasst mehr als 20.000 Zeilen Programmtext, der von uns konstruierte Checker nur ungefähr die Hälfte. Bereits daran sieht man, dass wir den Verifikationsaufwand deutlich verringern konnten. Wenn man außerdem bedenkt, dass im Checker die fehleranfällige und aufwändig zu verifizierende Suche nach einer optimalen bzw. guten Lösung *nicht* enthalten ist, erkennt man, dass wir mit der Methode des Programmprüfens mit Zertifikaten den Verifikationsaufwand deutlich verringern konnten.

7 Verwandte Arbeiten

Frühe Arbeiten zur Verifikation von Compilern (bei der Übersetzung der Programmiersprache Piton) wurden im Boyer-Moore-Beweiser durchgeführt [Moo89]. Das von der Deutschen Forschungsgemeinschaft geförderte *Verifix*-Projekt, das an den Universitäten in Karlsruhe, Kiel und Ulm durchgeführt wurde, hat Methoden entwickelt, mit denen korrekte Compiler konstruiert werden können, ohne dass Leistungseinbußen entstehen, siehe [GGZ04] für einen Überblick. Einige neuere Arbeiten wie zum Beispiel die Verifikation der Übersetzung von Java in Java Byte Code [KN03] haben sich auf die Übersetzungen im Frontend-Bereich konzentriert. Der Ansatz von beweistragendem Code (*proof-carrying code*) [Nec97] ist schwächer als der von uns verfolgte, weil er sich auf die Verifikation von notwendigen, nicht aber hinreichenden Korrektheitsbedingungen konzentriert. Programmprüfung wurde im Kontext algebraischer Probleme entwickelt [BK95] und wird nicht nur bei der Verifikation von Compilern, sondern auch bei der Verifikation von Bibliotheksroutinen [MN98] und der Hardware-Verifikation [GD04] eingesetzt.

8 Zusammenfassung und Ausblick

Wir haben in diesem Artikel einen Ausschnitt unserer Arbeiten vorgestellt und dabei nicht nur konkrete Ergebnisse zusammengefasst, die die Verifikation optimierender Compiler betreffen, sondern damit auch exemplarisch typische Vorgehensweisen in Forschungsarbeiten im Bereich Sicherheit erläutert. Bei der Verifikation optimierender Compiler sind sowohl abstrakt-logische als auch softwaretechnische Probleme zu lösen. Zum einen muss verifiziert werden, dass die angewandten Transformationsalgorithmen die Semantik der übersetzten Programme erhalten (*Übersetzungskorrektheit*). Zum anderen muss sichergestellt werden, dass diese Algorithmen in einem Compiler auch korrekt implementiert sind (*Implementierungskorrektheit*).

Diese Kombination von Anforderungen, theoretisch sinnvolle Problembeschreibungen und -lösungen zu finden, die in effizienten zuverlässigen Systemen implementiert werden, ist typisch für die Informatik. Mit unseren Ergebnissen tragen wir nicht nur dazu bei, optimierende Compiler, die ein wichtiges Werkzeug in der Softwaretechnik sind, zu verifizieren, sondern entwickeln auch Methoden, die allgemein bei der Transformation von Hardware- und Software-Systemen verwendbar sind. In unseren zukünftigen Arbeiten planen wir, diese Methoden nicht nur im Compilerbau, sondern auch allgemein bei dem Entwurf und der Implementierung sicherer und effizienter Systeme, insbesondere eingebetteter Systeme, einzusetzen und weiter zu entwickeln.

Danksagung: Mein Dank gilt Prof. Dr. Gerhard Goos und Prof. Dr. Wolf Zimmermann für die fruchtbare Zusammenarbeit und die zahlreichen Diskussionen, die zu den hier vorgestellten Ergebnissen beigetragen haben. Weiterhin gilt mein Dank den Mitgliedern meiner Arbeitsgruppe, insbesondere meinen Doktoranden Jan Olaf Blech und Lars Gesellensetter sowie den Studenten, die im Rahmen ihrer Studien- und Diplomarbeiten sowie als wissenschaftliche Hilfskräfte bei mir tätig sind und waren, für die erfolgreiche gemeinsame Arbeit. Schließlich möchte ich mich bei meinen Geldgebern, der Deutschen Forschungsgemeinschaft (Projektförderung unter dem Geschäftszeichen Gl 360/1-1) sowie der Landesstiftung Baden-Württemberg, für die Förderung bedanken, die mir früh in meiner wissenschaftlichen Laufbahn finanzielle Unabhängigkeit gegeben und damit eigenständige Forschung ermöglicht hat.

Literatur

[ASU92] Aho, A.V., R. Sethi und J.D. Ullman: *Compilerbau (Teil 1 und Teil 2)*. Addison-Wesley, 1992.

[AZ04] Aigner, Martin und Günter M. Ziegler: *Proofs from THE BOOK*. Springer-Verlag, 2004.

[BG04] Blech, Jan Olaf und Sabine Glesner: *A Formal Correctness Proof for Code Generation from SSA Form in Isabelle/HOL.* In: *Proceedings der 3. Arbeitstagung Programmiersprachen (ATPS) auf der 34. Jahrestagung der Gesellschaft für Informatik.* Lecture Notes in Informatics, September 2004.

[BGLM05] Blech, Jan Olaf, Sabine Glesner, Johannes Leitner und Steffen Mülling: *Optimizing Code Generation from SSA Form: A Comparison Between Two Formal Correctness Proofs in Isabelle/HOL.* In: *Proceedings of the Workshop Compiler Optimization meets Compiler Verification (COCV 2005), 8th European Conferences on Theory and Practice of Software (ETAPS 2005)*, Edinburgh, UK, April 2005. Elsevier, Electronic Notes in Theoretical Computer Science (ENTCS).

[BK95] Blum, Manuel und Sampath Kannan: *Designing Programs that Check Their Work.* Journal of the ACM, 42(1):269–291, 1995.

[Bor02] Borland/Inprise: *Official Borland/Inprise Delphi-5 Compiler Bug List.* http://www.borland.com/devsupport/delphi/fixes/delphi5/compiler.html, 2002.

[CM86] Chirica, Laurian M. und David F. Martin: *Toward Compiler Implementation Correctness Proofs.* ACM Transactions on Programming Languages and Systems, 8(2):185–214, April 1986.

[CV98] Crow, Judith und Ben Di Vito: *Formalizing Space Shuttle Software Requirements: Four Case Studies.* ACM Trans. Softw. Eng. Methodol., 7(3):296–332, 1998.

[DvHG03] Dold, Axel, Friedrich W. von Henke und Wolfgang Goerigk: *A Completely Verified Realistic Bootstrap Compiler.* International Journal of Foundations of Computer Science, 14(4):659–680, 2003.

[GB05] Glesner, Sabine und Jan Olaf Blech: *Logische und softwaretechnische Herausforderungen bei der Verifikation optimierender Compiler.* In: *Proceedings der Software Engineering 2005 Tagung (SE 2005).* Lecture Notes in Informatics, März 2005.

[GD04] Grosse, Daniel und Rolf Drechsler: *Checkers for SystemC Designs.* In: *Proceedings of the Second ACM & IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE'2004)*, Seiten 171–178, San Diego, USA, 2004.

[GFJ04] Glesner, Sabine, Simone Forster und Matthias Jäger: *A Program Result Checker for the Lexical Analysis of the GNU C Compiler.* In: *Proceedings of the Workshop Compiler Optimization meets Compiler Verification (COCV 2004), 7th European Conferences on Theory and Practice of Software (ETAPS 2004)*, 2004. Elsevier, Electronic Notes in Theoretical Computer Science (ENTCS).

[GGZ98] Goerigk, Wolfgang, Thilo Gaul und Wolf Zimmermann: *Correct Programs without Proof? On Checker-Based Program Verification.* In: *Tool Support for System Specification and Verification, ATOOLS'98*, Malente, Germany, 1998. Springer Series Advances in Computing Science.

[GGZ04] Glesner, Sabine, Gerhard Goos und Wolf Zimmermann: *Verifix: Konstruktion und Architektur verifizierender Übersetzer (Verifix: Construction and Architecture of Verifying Compilers).* it - Information Technology, 46:265–276, 2004. Print ISSN: 1611-2776.

[Gle03a] Glesner, Sabine: *Program Checking with Certificates: Separating Correctness-Critical Code.* In: *Proceedings of the 12th International FME Symposium (Formal Methods Europe)*, Seiten 758–777, Pisa, Italy, September 2003. Springer Verlag, Lecture Notes in Computer Science, Vol. 2805.

[Gle03b] Glesner, Sabine: *Using Program Checking to Ensure the Correctness of Compiler Implementations.* Journal of Universal Computer Science (J.UCS), 9(3):191–222, March 2003.

[Gle04a] Glesner, Sabine: *A Proof Calculus for Natural Semantics Based on Greatest Fixed Point Semantics.* In: *Proceedings of the Workshop Compiler Optimization meets Compiler Verification (COCV 2004), 7th European Conferences on Theory and Practice of Software (ETAPS 2004)*, Barcelona, Spain, April 2004. Elsevier, Electronic Notes in Theoretical Computer Science (ENTCS).

[Gle04b] Glesner, Sabine: *An ASM Semantics for SSA Intermediate Representations.* In: *Proceedings of the 11th International Workshop on Abstract State Machines*, Halle, Germany, May 2004. Springer Verlag, Lecture Notes in Computer Science.

[Gle04c] Glesner, Sabine: *Verification of Optimizing Compilers*, Habilitationsschrift (Habilitation Thesis), Fakultät für Informatik, Universität Karlsruhe. November 2004.

[Har00] Harrison, John: *Formal verification of IA-64 division algorithms.* In: *Theorem Proving in Higher-Order Logics: 13th International Conference, TPHOLs 2000*, Seiten 234–251. Springer Verlag, Lecture Notes in Computer Science, Vol. 1869, 2000.

[HGG+99] Heberle, Andreas, Thilo Gaul, Wolfgang Goerigk, Gerhard Goos und Wolf Zimmermann: *Construction of Verified Compiler Front-Ends with Program-Checking.* In: *Perspectives of System Informatics, Third International Andrei Ershov Memorial Conference, PSI'99*, Akademgorodok, Novosibirsk, Russia, July 1999. Springer, Lecture Notes in Computer Science, Vol. 1755.

[JR97] Jacobs, Bart und Jan Rutten: *A Tutorial on (Co)Algebras and (Co)Induction.* EATCS Bulletin, 67:222–259, 1997.

[KN03] Klein, Gerwin und Tobias Nipkow: *Verified Bytecode Verifiers.* Theoretical Computer Science, 298:583–626, 2003.

[MN98] Mehlhorn, Kurt und Stefan Näher: *From Algorithms to Working Programs: On the Use of Program Checking in LEDA.* In: *23rd International Symposium on Mathematical Foundations of Computer Science (MFCS'98)*, 1998. Springer, Lecture Notes in Computer Science, Vol. 1450.

[Moo89] Moore, J. S.: *A Mechanically Verified Language Implementation.* Journal of Automated Reasoning, 5(4):461–492, 1989.

[MP67] McCarthy, John und J. Painter: *Correctness of a Compiler for Arithmetic Expressions.* In: *Mathematical Aspects of Computer Science, Proceedings of Symposia in Applied Mathematics*, Seiten 33–41. American Mathematical Society, 1967.

[Muc97] Muchnick, Steven S.: *Compiler Design and Implementation.* Morgan Kaufmann Publishers, 1997.

[Nec97] Necula, George C.: *Proof-Carrying Code.* In: *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'97)*, 1997.

[New01] Newsticker, Heise: *Rotstich durch Fehler in Intels C++ Compiler.* http://www.heise.de/newsticker/data/hes-11.11.01-000/, 2001.

[NNH99] Nielson, Flemming, Hanne Riis Nielson und Chris Hankin: *Principles of Program Analysis.* Springer, 1999.

[NPW02] Nipkow, Tobias, Lawrence C. Paulson und Markus Wenzel: *Isabelle/HOL: A Proof Assistant for Higher-Order Logic.* Springer, Lecture Notes in Computer Science, Vol. 2283, 2002.

[NvO98] Nipkow, Tobias und David von Oheimb: Java$_{light}$ *is Type-Safe – Definitely.* In: *Proceedings of the 25th ACM Symposium on the Principles of Programming Languages*, 1998. ACM Press.

[OST+02] Ortmeier, F., G. Schellhorn, A. Thums, W. Reif, B. Hering und H. Trappschuh: *Safety Analysis of the Height Control System of the Elbtunnel.* In: *Computer Safety, Reliability and Security, 21st Int'l Conference, SAFECOMP 2002.* Springer, Lecture Notes in Comp. Sc., Vol. 2434, 2002.

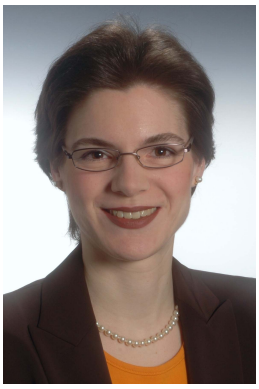
[Pap94] Papadimitriou, Christos H.: *Computational Complexity.* Addison-Wesley, 1994.

[Pol81] Polak, Wolfgang: *Compiler Specification and Verification.* Springer, Lecture Notes in Computer Science, Vol. 124, 1981.

[PSS98] Pnueli, A., M. Siegel und E. Singerman: *Translation validation.* In: *Proceedings of Tools and Algorithms for the Construction and Analysis of Systems*, Seiten 151–166, Lisbon, Portugal, April 1998. Springer Verlag, Lecture Notes in Computer Science, Vol. 1384.

[SA97] Schellhorn, Gerhard und Wolfgang Ahrendt: *Reasoning about Abstract State Machines: The WAM Case Study.* Journal of Universal Computer Science, 3(4):377–413, 1997.

Über die Autorin



PD Dr. Sabine Glesner hat 1996 an der Technischen Universität Darmstadt ihr Informatikstudium "mit Auszeichung" abgeschlossen. Zuvor hatte sie mit einem Fulbright-Stipendium den Master of Science in Computer Science an der University of California, Berkeley erworben. Während ihres Studiums war Dr. Glesner Stipendiatin der Studienstiftung des deutschen Volkes. Ihre Promotion führte sie von 1996 bis 1999 in der Fakultät für Informatik an der Universität Karlsruhe durch. Für ihre Dissertation, die sie "mit Auszeichnung" abgeschlossen hat, erhielt sie den Förderpreis des Forschungszentrums Informatik. Sabine Glesner ist als Gutachter für internationale Konferenzen und Zeitschriften sowie als Mitglied in Programmkomitees internationaler Workshops tätig. Ihre aktuellen Forschungsarbeiten werden durch das Margarete von Wrangell-Habilitationsprogramm des Landes Baden-Württemberg, durch das Eliteförderprogramm für Postdoktoranden der Landesstiftung Baden-Württemberg sowie durch den Aktionsplan Informatik der Deutschen Forschungsgemeinschaft (DFG) im Rahmen des Emmy Noether-Programms gefördert, der ihr den Aufbau ihrer Arbeitsgruppe ermöglicht hat. Sabine Glesner hat ihre Habilitation im Mai 2005 an der Universität Karlsruhe abgeschlossen und die Lehrbefugnis für das Fach Informatik erhalten. Im August 2005 hat sie einen Ruf auf eine Professur für Programmierung eingebetteter Systeme an der Technischen Universität Berlin angenommen und einen weiteren Ruf auf eine Professur für Theorie der Programmiersprachen und Programmierung an der Universität Rostock abgelehnt.